Malloc Report

Operating Systems

Osbaldo Puente Cerda

University of Texas at Arlington

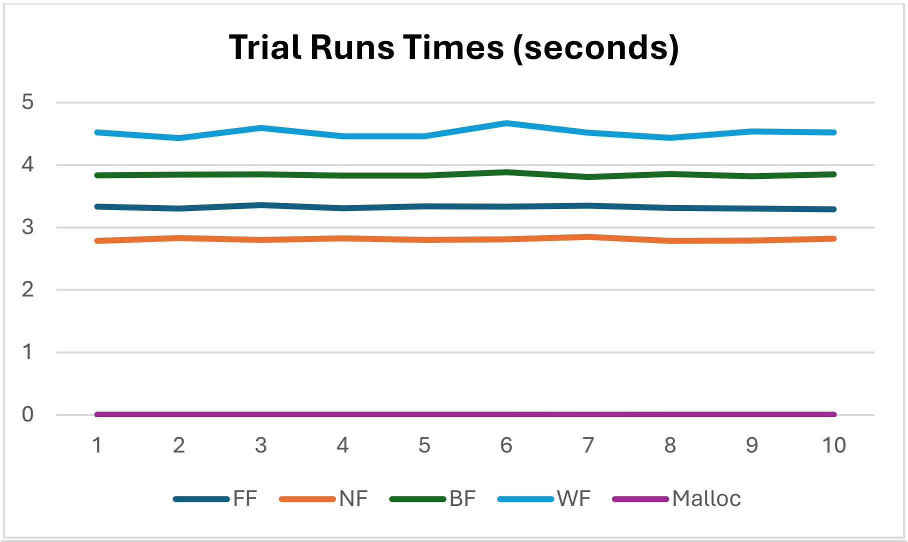
CSE-3320-001

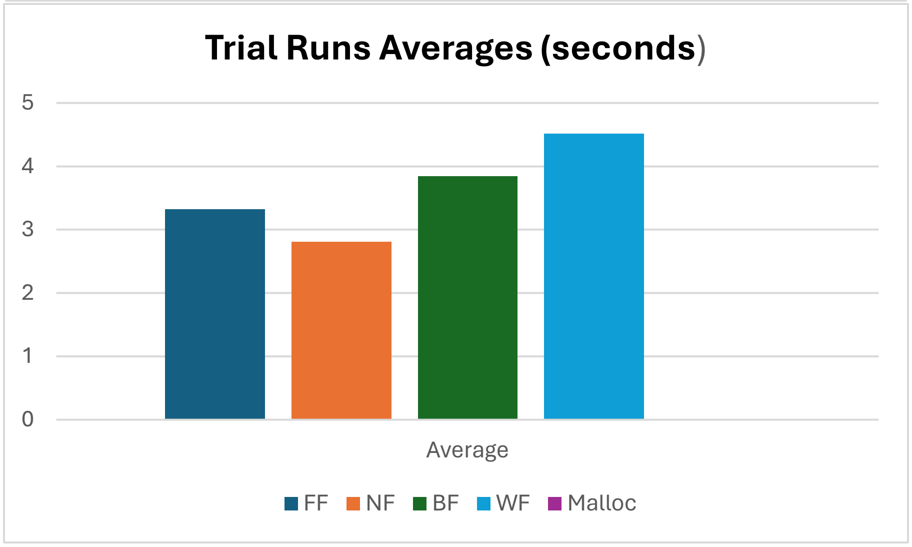
Time Statistics

Time Recorded (seconds)

Algorithm

Trial #	FF	NF	BF	WF	Malloc
1	3.333	2.788	3.837	4.522	0.004387
2	3.301	2.832	3.847	4.431	0.004572
3	3.359	2.799	3.85	4.595	0.004441
4	3.306	2.827	3.833	4.462	0.005072
5	3.34	2.803	3.831	4.459	0.004364
6	3.336	2.813	3.885	4.67	0.004249
7	3.349	2.85	3.808	4.518	0.00422
8	3.314	2.786	3.856	4.433	0.004474
9	3.301	2.793	3.818	4.538	0.004237
10	3.291	2.82	3.853	4.521	0.004419
Average	3.323	2.8111	3.8418	4.5149	0.004444

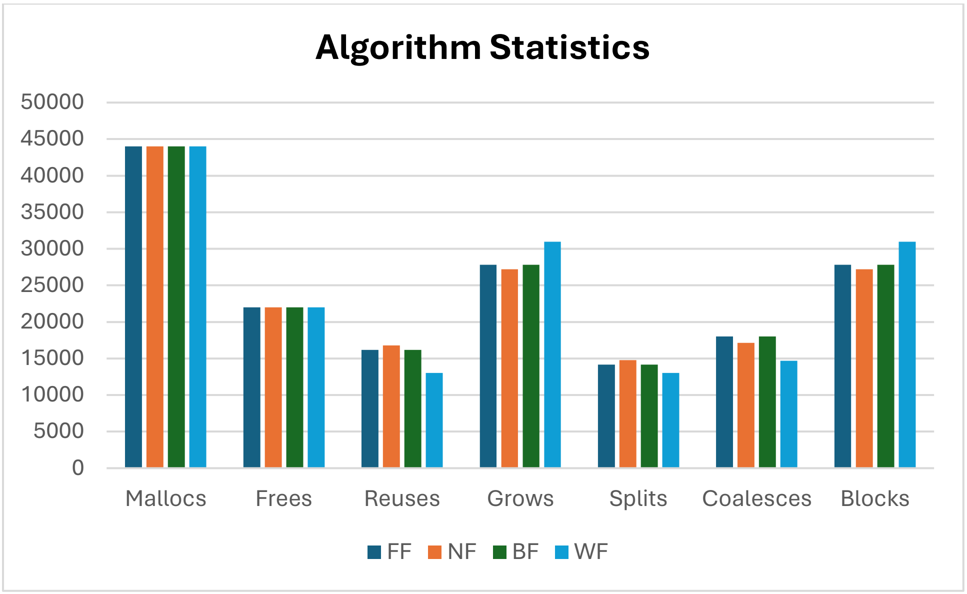


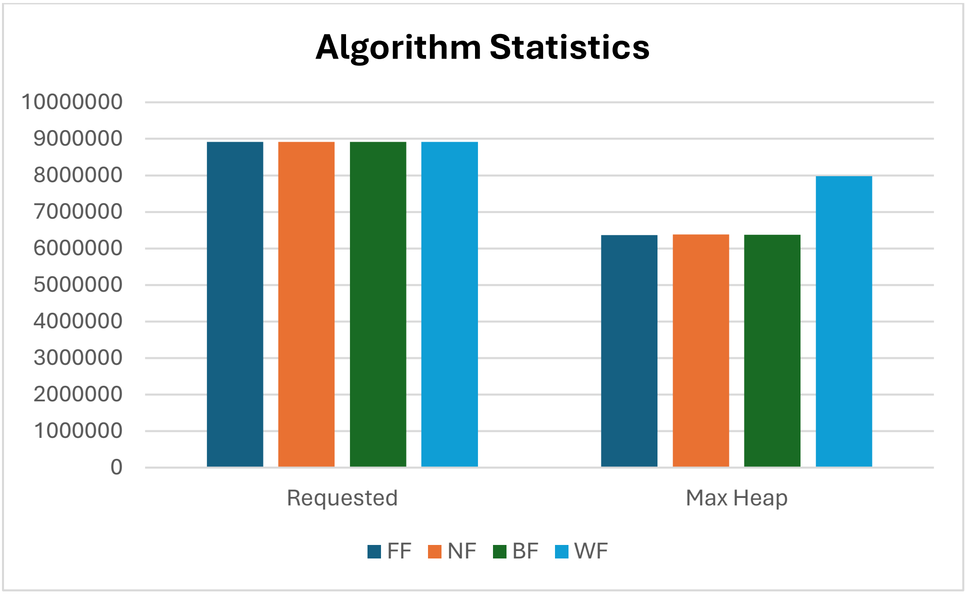


Algorithm Statistics

Algorithm Statistics

Statistic	FF	NF	BF	WF
Mallocs	44007	44007	44007	44007
Frees	22017	22017	22017	22017
Reuses	16175	16793	16168	13034
Grows	27832	27214	27839	30973
Splits	14159	14777	14153	13016
Coalesces	18006	17145	17998	14711
Blocks	27832	27214	27839	30973
Requested	8916808	8916808	8916808	8916808
Max Heap	6366496	6381088	6369584	7974304





Executive Summary

The following report implements 4 memory allocation algorithms. The algorithms are called first fit, next fit, best fit, and worst fit. They are tested based on their time performance along with tracking certain statistics. The statistics tracked are the number of mallocs called, the number of blocks feed, the number of blocks reused, the number of times the heap grows, the number of times a block is split for being larger than needed, the number of times blocks coalesce, the number of blocks created, the total amount of memory requested, and the maximum size the heap grows.

Description of Algorithms

This report runs and tests 4 algorithms along with the standard systems malloc() function against each other to compare their performance. The 4 algorithms are as follows.

First Fit: Iterates through the heap starting from the beginning and finds the first block that is large enough for the size that has been requested.

Next Fit: Iterates through the heap starting from the last block that was last used. From there it finds the next block that is large enough for the size that has been requested. If it reaches the end of the heap, it will loop back to the top of the heap and iterate from there until its original starting position.

Best Fit: Iterates through the heap starting from the top until the end and uses the block that is filled the most by the size requested.

Worst Fit: Iterates through the heap starting from the top until the end and uses the block which is filled the least by the size requested.

Test Implementation

My test program, included in the appendix, aims to test how the different algorithms will execute the same tasks. The program aims to force the algorithms to split and coalesce multiple blocks. This is most easily seen with the code that used array3. The code initializes an array for a certain size, it then frees only certain pointers in the array. This is to test how each algorithm will iterate through when next using malloc. The code then reinitialized the pointers to a smaller size attempting to force the algorithms to split the previously freed blocks. The code then does the opposite and reinitializes certain pointers to a larger size attempting to force the algorithms to grow the heap.

Explanation and Interpretation

Time Statistics

Looking at the trial runs we see that the systems malloc function performed the fastest, which is to be expected, with an average run time of 0.004 seconds. After malloc, in order from fastest to slowest, we have next fit, first fit, best fit, and lastly worst fit. There is a significant difference between the next fit and worst fit algorithms with nearly a 60% decrease in performance speed.

Algorithm Statistics

The results for the number of mallocs, the number of frees, and the total memory requested stayed consistent throughout all four algorithms. This is expected as they should all perform these functions the same way. We do see some differences in the rest of the statistics recorded though.

Looking at the number of grows, the worst fit algorithm grows the heap more than any other algorithm. It grows 3759 times more than the next fit algorithm, which is the algorithm that grew the heap the least. This is in line with the time performance for both algorithms and seems to be part of the reason the worst fit algorithm performed a lot slower. The number of reuses for each algorithm also correlates with the data for the number of grows as the next fit algorithm has the highest number of reuses while the worst fit algorithm has the lowest number of reuses. Along with this, the number of grows causes the maximum size of the heap to be greater. This is shown in our data where the max heap of worst fit is an outlier amongst the 4 algorithms being significantly larger.

What is interesting though is the number of splits for each algorithm. I would have expected the worst fit algorithm to have the highest number of splits as it searches the heap for the block that has the most remaining space. In this test case, however, the next fit algorithm had the highest number of splits. In our test case, the first fit algorithm had the highest number of coalesces. This is likely due to the first fit algorithm normally causing more fragmentation among blocks.

Conclusion

Based on the results of the report, the best algorithm to use based on time performance would be the next fit algorithm for this test program. The next fit algorithm had a good balance of speed and was effective in this case due to the minimal block fragmentations. However, the choice of which algorithm to utilize should be considered based on the requirement and code of the program being tested. This test highlights the trade-off between performance and certain functions being performed.

Test Program

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int main(){

    clock_t start, end;
    double time;
    start = clock();
    printf("Starting Program Testing\n");

    int arraySize = 2000;
    char *array[arraySize];

    for(int i=0; i<arraySize; i++){
        array[i] = (char*)malloc(512);
    }

    char *ptr1 = (char*)malloc(4096);

    for(int i=1; i<arraySize; i=i*2){
        free(array[i]);
    }

    free(ptr1);

    char *ptr2 = (char*)malloc(4096);
    char *ptr = (char*)malloc(4);

    free(ptr2);

    char *array2[arraySize];
    for(int j=0; j<arraySize; j++){
        array2[j] = (char*)malloc(16);
    }

    char *ptr3 = (char*)malloc(131072);
    free(ptr3);
    free(ptr);

    for(int j=0; j<arraySize; j++){
        free(array2[j]);
    }
    char *ptr4 = (char*)malloc(4);
    free(ptr4);

    int bigggerArraySize = arraySize *10;
    char *array3[bigggerArraySize];
    for(int j=0; j<bigggerArraySize; j++){
        array3[j] = (char*)malloc(128);
    }

    free(array3[bigggerArraySize-1]);
    array3[bigggerArraySize-1] = (char*)malloc(512);

    for(int j=0; j<bigggerArraySize; j++){
        if(j%2 == 0){
            free(array3[j]);
        }
    }

    for(int j=0; j<bigggerArraySize; j++){
        if(j%2 == 0){
            array3[j] = (char*)malloc(4);
        }
    }

    for(int j=0; j<bigggerArraySize; j++){
        if(j%2 != 0){
            free(array3[j]);
        }
    }

    for(int j=0; j<bigggerArraySize; j++){
        if(j%2 != 0){
            array3[j] = (char*)malloc(512);
        }
    }

    end = clock();
    time = (double)(end-start);
    printf("Time Elapsed: %f\n", (time/CLOCKS_PER_SEC));


    return 0;
}
```